Exploring the Role of Diagnosis in Software Failure

Les Hatton, *Oakwood Computing and the University of Kent, UK*

Among engineering systems, software systems are unique in their tendency toward repetitive failure, and the situation is worsening as systems become larger and more tightly coupled. To stem the tide, the author advocates a humbler attitude toward software failure and improved diagnostics.

When an engineering system behaves in an unexpected way, we say it has failed. If the failure is significant, we use a process called *diagnosis* to discover why the failure occurred. By definition, the successful outcome of diagnosis is the discovery of a deficiency whose correction prevents the system from failing in the same way again. We call such a deficiency a *fault*. If diagnosis is unsuccessful, the fault remains undiscovered and can cause the system to fail again in the future. Such a failure is called a *repetitive failure* mode, although it would probably be clearer to call it a *diagnostic failure* mode.

The first main point of this article is that repetitive failure is very common in software-controlled systems as compared to other kinds of engineering systems. Second, I argue that this state of affairs stems from an overly optimistic approach by software system designers and implementers, coupled with rapidly increasing system complexity. Finally, in discussing ways to improve the situation, I define two attributes of diagnosis that may prove useful in clarifying the problem.

Evidence for repetitive failure

Repetitive failure is indeed a widespread property of software-controlled systems. For early evidence of this, we can go back more than 15 years, when Ed Adams published his now famous study at IBM,[1] showing that a very small percentage of faults—only around 2%—were responsible for most observed failures. In other words, his data showed that his systems were dominated by repetitive failure.

Figure 1 shows the relationship between fault and failure. In essence, every failure results from at least one fault, but not all faults fail during the software's life cycle. In fact, Adams showed that a third of all faults failed so rarely that for all intents and purposes they never failed at all in practice. It is quite easy to envisage a fault that does not fail. For example, large systems sometimes contain unintentionally unreachable code; a fault located in an unreachable piece of code could never fail. In general, however, under some set of conditions, a fault or group of faults in combination will cause the system to fail.

Figure 2 presents an analysis of dependencies on a significant class of software faults oc-

0740-7459/01/$10.00 © 2001 IEEE

curring in a large study of commercial C systems.[2] In this case, I calculated the weighted fault rate by counting the occurrences of 100 of C's better-known fault modes and weighting them according to severity between 0 and 1.[3] This class of faults is widely published, has been known for the best part of 10 (and in some cases 20) years, is known to fail, and yet nothing much has been done to prevent it. Precisely the same thing occurs to differing extents for other standardized languages.

It is perhaps no wonder that independently written programs tend to suffer from nonindependent failure,[4] when software in general is riddled with a growing number of known repetitive failure modes that we seem unable to avoid.

Reasons for repetitive failure

Why would software systems be dominated by repetitive failure? After all, other engineering systems aren't. If bridges failed in the same way repeatedly, there would be a public outcry. The answer seems to lie in maturity: If you go far enough back in time, bridges *did* fail repeatedly, and there *was* public outcry. For example, in 1879, the Tay Bridge in Scotland fell down in a strong gale. It was poorly designed and built, and the wind blew it over, causing the deaths of 75 people in the train that was on it at the time. More or less as a direct result of the ensuing public outcry, the Forth Bridge in Scotland, built a little later, is almost embarrassingly overengineered. If a comet ever hits the earth, let's hope that it hits the Forth Bridge first. It will very likely bounce off.

In fact, in the 1850s, as many as one in four iron railway bridges fell down until the reason was discovered.[5] The process of engineering maturation, whereby later designs gradually avoided past mistakes, has led to the almost complete disappearance of repetitive failure—not only in bridges, but also in most other areas of engineering. The single exception is software. It is easy to think that this is because software development is only around 50 years old, but this is overly generous and ignores the lessons of history. The fact is that software engineering is gripped by unconstrained and very rapid creativity, whereas the elimination of repetitive failure requires painstaking analysis of relatively slowly moving technologies.

This standard method of engineering improvement, summarized in Figure 3, has been known for a very long time. It used to be called common sense, but in these enlightened times, with the addition of a little mathematics, we know it as *control process feedback*.

Regrettably, use of this simple principle is not widespread in software engineering, although process control models such as the SEI CMM firmly espouse it. It is almost as if the idea is too obvious, a drawback that also

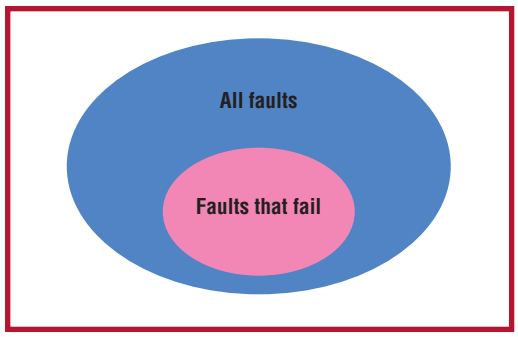


Figure 1. The relationship between fault and failure. There is not necessarily a one-to-one relationship between failure and "faults that fail." A failure is simply a difference between a system's expected behavior and its actual behavior; some failures result from two or more faults acting collaboratively.

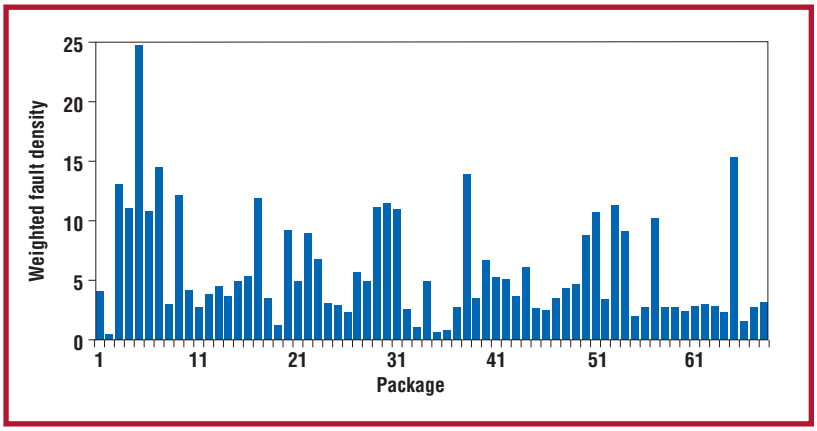


Figure 2. Weighted fault rates per 1,000 lines of code for a wide variety of commercially released C applications, plotted as a function of package number. This study analyzed 68 packages (totaling over 2,000,000 nonblank, preprocessed lines) from approximately 50 application areas in approximately 30 industrial groupings. The groupings ranged from non–safety-related areas, such as advertising and insurance, to safety-critical areas, such as air-traffic control and medical systems. Safety-critical systems were a little better than the average (which was 5.73), but one of the games measured scored better than two of the medical systems.

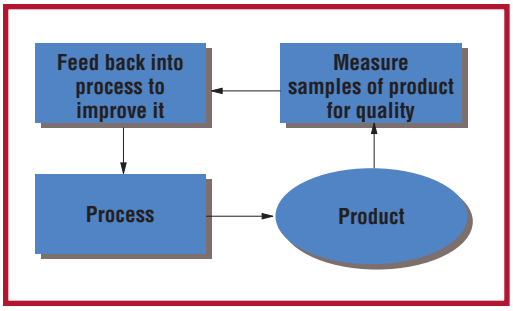


Figure 3. The silver bullet of engineering, control process feedback embodies the extraordinarily simple principle that it is not a sin to make a mistake, it is a sin to repeat one. Careful analysis of failure reveals the all-important clues as to how to avoid it in the future. The analysis mechanism is known as root-cause analysis.

might explain attitudes over the last 200 years or so toward Darwinian evolution.

The writer and technological seer Arthur C. Clarke once said that "any sufficiently complex technology is indistinguishable from magic." I would add that any sufficiently *simple* technology is also indistinguishable from magic. Darwinian evolution and control process feedback are two outstanding examples of my principle in action. They are apparently so obvious that they shouldn't be right. However, probably the most important characteristic these two share is that although improvement is inexorable, it also takes time. This is particularly relevant in these days of "reduced time to market"—perhaps better known as "don't test it as much."

Another demonstrable source of repetitive failure in software systems is imprecisely defined programming languages—a problem that many organizations make no effort to avoid. The language standardization process exacerbates this problem. Obviously, standardization is an important step forward in engineering maturity, but the process should not ignore historical lessons. As practiced today, language standardization suffers from two important drawbacks.

First, language committees (and I've sat on a few in my time) seem unable to resist the temptation to fiddle, albeit with every good intention. These committees add features that seem like a good idea at the time, but often without really knowing whether they will work in practice. Of course, such creativity is normal in engineering. It is similar to the role of mutation in Darwinian evolution.

What is not normal, however, is language standardization's second drawback, the concept of *backwards compatibility,* often expressed in the hallowed rule, "Thou shalt not break old code." The backwards compatibility principle operates in direct opposition to control process feedback.

So, drawback one guarantees the continual injection of features that may not work, and drawback two guarantees the extreme difficulty of taking them out again. In other words, these two characteristics guarantee a standardization technique that largely excludes learning from previous mistakes. If other engineering disciplines pursued this doctrine, hammers, for example, would have microprocessor-controlled ejection mechanisms to ensure that their heads would fly off randomly every few minutes—just as they did when made with wooden handles 40 years ago. Not surprisingly, hammers were redesigned to eliminate this feature. We can clearly see the effects of the language situation in Figure 2.

A further reason that software is full of repetitive failure modes is my main subject in this article—the diagnosis problem. If a system fails and diagnosis does not reveal the fault or all of the faults that caused the failure (bearing in mind that some experiments have shown that one in seven defect corrections is itself faulty![1]), the failure will occur again in some form in the future. In other words, the inability to diagnose a fault inevitably results in the resulting failure becoming repetitive.

Now I'll attempt to analyze why diagnosis often fails.

Quantifying diagnosis

The essence of diagnosis is to trace back from a failure to the culpable fault or faults. Unfortunately, we have no systematic model that lets us go in the other direction—that is, to predict failure from a particular fault or group of faults. This is the province of the discipline known as "software metrics," wherein we attempt to infer runtime behavior and, in particular, failure occurrence from some statically measurable property of the code or design. For example, we might say that components with a large number of decisions are unusually prone to failure. In general, we have so far failed to produce any really useful relationship, and the area is very complex.[6] The primary reason for this appears to be that software failure is fundamentally chaotic. In conventional engineering systems, we have learned through hard experience to work in the linear zone, where stress is linearly related to strain and the the system's behavior at runtime is much more predictable, although still occasionally prone to chaotic failure.[5] Unfortunately, at the present stage of understanding, software is much less predictable.

```
Dereference pointer content 0x0 at
strlen(…) called from
line 126 of myc_constexpr.c called from
line 247 of myc_evalexpr.c called from
line 2459 of myc_expr.c
```

Figure 4. A typical stack trace automatically produced by a reasonable C compiler (in this case the estimable GNU compiler), at the point of dereferencing a pointer containing the address zero.

This asymmetry between prediction and diagnosis is particularly clear when we consider the difference between code inspections and traditional testing. Code inspections address the whole fault space of Figure 1. Consequently, a large percentage of faults revealed during code inspections would never actually cause the system to fail in a reasonable life cycle. In spite of this, the often dramatic effectiveness of code inspections compared with traditional runtime testing is unquestionable.[7] Traditional testing, of course, finds a problem at runtime. This by definition is a failure and lies in the small subset of Figure 1. (In fact, we can predict that code inspections will be even more effective on systems that rapidly build up many execution years, such as modern consumer embedded systems, because in these cases the subset of faults that fail is much bigger.)

Two essential parameters help us categorize how easily a software failure can be diagnosed. These are *diagnostic distance* and *diagnostic quality*.

Diagnostic distance

In essence, diagnostic distance is the "distance" in the system state between a fault being executed and the resulting failure manifesting itself. We can reasonably visualize this distance as the number of changes of state (for example, variables or files modified) that occur between the fault's execution and the observation of failure. The greater the number of changes, the more difficult in general it is to trace the failure back to the fault—that is, to diagnose it. Let's consider some simple examples from different languages and different systems.

Consider Figure 4, a typical "stack trace" resulting from a core dump, generated in this case by a C program. The program has tried to look at the contents of address zero,

```
…
if (tolerance.eq. acceptable_tolerance) then
…
```

Figure 5. A comparison of real-valued variables. This is a broken concept in just about every programming language in existence and is one of the features included in the study that led to the data of Figure 2.

a fatal mistake in C leading to an immediate program failure. In other words, the "distance" between the fault firing and the failure occurring is very short. This, coupled with the detailed nature of the stack trace, points unerringly back at the precise location of the fault. Not surprisingly, these failures are very easy to diagnose.

We can contrast this failure with the next one, shown in Figure 5, which affected me some years ago. This failure occurred in a Fortran 77 program, but exactly the same problem manifests itself in other languages without warning today. In this case, a comparison of real variables behaved slightly differently on different machines, a common problem. The effect was an unacceptable drop in the significance of agreement between two different computers—from around four decimal places to only two decimal places—in some parts of a data set that formed part of an acceptance test. This line of code was buried in the middle of a 70,000+-line package containing various signal-processing algorithms. From the point of failure, a colleague and I spent some three months on-and-off tracing the problem back to this line of code.

In this case, the diagnostic distance between the fault and the failure was sufficiently large to lead to an exceptionally difficult debugging problem—although in the rich glow of hindsight, it is statically detectable.

We can see other examples of large diagnostic distance in dynamic-memory failures in mainstream languages such as C, C++, and Fortran, to a lesser extent in Ada, and also in several aspects of OO implementations. In these cases, dynamic memory is allocated, corrupted at some stage, and rather later on leads to a failure. We can also call this kind of failure *nonlocal behavior*.

Diagnostic quality

In contrast to diagnostic distance, diagnostic quality refers to how well the diagnostic distance is signposted between the state at which the fault executed and the state at which we observe the software to fail. To give a simple example of how influential this can be, remove a significant token from the middle of a computer program, for example, a "," from a C program. When

```
…
myc_decl.c:297 parse error before 'name'
…
(lots of increasingly more exotic messages)
…
myc_decl.c: 313: warning: data definition
has no type or storage class
```

Figure 6. Part of the fault cascade on one of the author's programs, which has had a significant token, in this case a "," deliberately removed. Warnings near the point of occurrence are relatively easy to understand. Warnings later in the cascade rapidly become more and more esoteric.

such a program is recompiled, the compiler will usually diagnose the absence of this token with a reasonably comprehensible error message. Figure 6 shows an example from the GNU compiler.

Note that the missing token leads to a cascade of error messages that start out reasonably comprehensible at the fault location but rapidly degenerate into entirely spurious messages, until the compiler eventually gives up in disgust. This is a classic problem in compiler design for all programming languages. Now imagine trying to diagnose the missing token from the last compiler warning. Clearly, this can be very difficult. The full error cascade signposts the fault-failure path. Removing part of the signposting can fatally cripple your ability to diagnose the fault or faults from the failure.

Such cascades are becoming common in real systems as they become more tightly coupled and larger. Consider the example in Figure 7, quoted by Peter Mellor[8] in an excellent discussion of this topic.

Imagine trying to diagnose the landing gear problem from the last error message in this case! In fact, this led to a repetitive failure mode, as at least one more incident occurred (19 November 1988), and a further nine months went by before a fix could be made.

Training engineers to provide adequate signposting so that faults can be diagnosed from failures is an educational issue. We all too often fail to train software engineers to anticipate failure, and it is uncommon to teach the fundamental role of testing and techniques such as hazard analysis in our universities. This lack encourages overoptimism and leads to inadequate preparation for the inevitable failures. The diagnostic link between failure and fault is simply not present, and the fault becomes difficult, if not impossible, to find. Failure is one of the natural properties of a software-engineering system, and we ignore this immutable fact at our peril.

Even when diagnostic links are present during testing, engineers often remove them from or truncate them in the released system

- MAN PITCH TRIM ONLY, followed in quick succession by:
- Fault in right main landing gear;
- At 1,500 feet, fault in ELAC2, (one of seven computers in the Electronic Flight Control System);
- LAF alternate ground spoilers 1-2-3-5 (fault in Load Alleviation Function);
- Fault in left pitch control green hydraulic circuit;
- Loss of attitude protection (which prevents dangerous maneuvers);
- Fault in Air Data System 2;
- Autopilot 2 shown as engaged despite the fact that it was disengaged; and then, finally,
- LAVATORY SMOKE, indicating a (nonexisting) fire in the toilets.

Figure 7. The diagnostic sequence appearing on the primary flight display of an Airbus A320, Flight AF 914 on 25 August 1988. The faults were not spurious—the aircraft had difficulty getting its landing gear down and had to do three passes at low altitude by the control tower so that the controllers could check visually.

for space reasons, crippling our ability to diagnose failure properly when it occurs in the field. Consider the following examples of possible warnings issued in commercial systems.

Please wait ...

accompanied a failure in the flight management system on an Airbus A340 in September 1994.[9] To my knowledge, the cause has still not been found.

System over-stressed ...

appeared in a cash register system in a public bar. It later transpired that the printer had run out of paper.[10]

More than 64 TCP or UDP streams open ...

appeared in a G3 Macintosh running OS8.1. It turned out that the modem was not switched on.[10]

Of course, the ultimate in poor diagnostics is no warning at all. At the time of writing, there is considerable public debate in the UK about the Chinook helicopter crash in 1994 on the Mull of Kintyre in Scotland.[11] The crash killed 30 people, including both pilots and several very senior security people from Northern Ireland. Although the crash was

blamed on pilot error, prior to this incident concerns had been raised about the quality of the FADEC software controlling this aircraft's engines—including concerns about a design flaw that had precipitated the near destruction of another Chinook in 1989. The essence of the verdict in this case seems to be that because the crash investigators found no evidence that the FADEC software failed, it must have been the pilots. However, as any PC user knows, most PC crashes leave no trace, so that on reboot, there is no evidence that anything was ever wrong. It is therefore grievously wrong to equate no diagnosis with no failure.

Unification

We can put together diagnostic distance and diagnostic quality to summarize the diagnosis problem, as shown in Figure 8. Here, we can see that when diagnostic distance is considerable, unless diagnostic quality is good, we have a very difficult diagnostic problem, one that in general will be intractable. The "difficult" sector of Figure 8 can be seen as a source of repetitive failure. We can ameliorate such a problem either by improving the diagnostic quality or by reducing the distance between the point where the fault executed and the point where the failure was observed. In practice, the latter is much easier.

In the last few years, software engineering has begun producing more and more systems where diagnostic distance is large. Networking, for example, is a classic method of increasing diagnostic distance. Embedded systems are similarly difficult to diagnose. Unless we can make rapid progress in improving diagnostic quality—essentially an educational issue—things are going to spiral out of control, and we can look forward to repetitive failure becoming a permanent fixture in software-engineering systems. This is clearly an unacceptable scenario.

We can address both these issues by educating software engineers to realize that failure is an inevitable—indeed, a natural property of software systems—and to reflect this fact in design, implementation, and provision for a priori diagnosis.

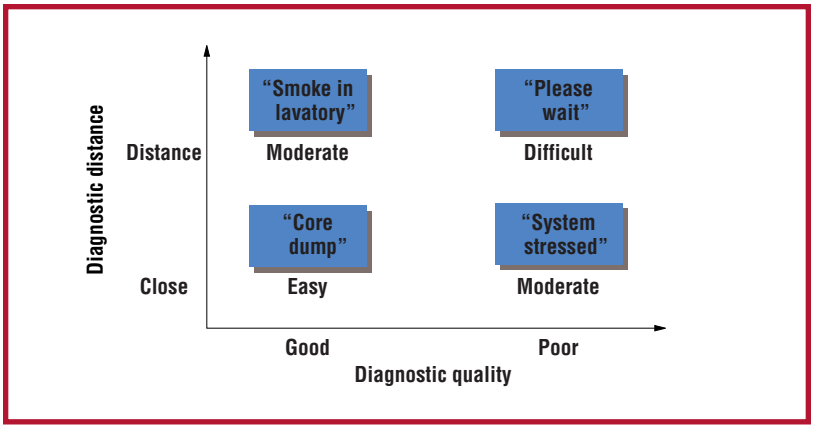


Figure 8. Diagnostic distance versus diagnostic quality. Failures discussed earlier in the article give examples of the four sectors.

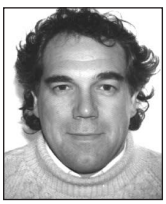
Acknowledgments

All sorts of people contributed to this article. In particular, I thank Peter Mellor for his input and the anonymous reviewers, who contributed enormously.

References

1. N.E. Adams, "Optimizing Preventive Service of Software Products," *IBM J. Research and Development*, vol. 28, no. 1, 1984, pp. 2–14.
2. L. Hatton, "The T Experiments: Errors in Scientific Software," *IEEE Computational Science & Eng.*, vol. 4, no. 2, Apr.–Jun. 1997, pp. 27–38.
3. L. Hatton, *Safer C: Developing for High-Integrity and Safety-Critical Systems*, McGraw-Hill, New York, 1995.
4. J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming," *IEEE Trans. Software Eng.*, vol. 12, no. 1, 1986, pp. 96–109.
5. H. Petroski, *To Engineer is Human: The Role of Failure in Successful Design*, Random House, New York, 1992.
6. N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd edition, PWS Publishing, Boston, 1997.
7. T. Gilb and D. Graham, *Software Inspections*, Addison-Wesley, Wokingham, England, 1993.
8. P. Mellor, "CAD: Computer-Aided Disaster," *High Integrity Systems*, vol. 1, no. 2, 1994, pp. 101–156.
9. *AAIB Bulletin 3/95*, Air Accident Investigation Branch, DRA Farnborough, UK, 1995.
10. L. Hatton, "Repetitive Failure, Feedback and the Lost Art of Diagnosis," *J. Systems and Software*, vol. 47, 1999, pp. 183–188.
11. A. Collins, "MPs Misled Over Chinook," *Computer Weekly*, 27 May 1999.

About the Author



Les Hatton is an independent consultant in software reliability. He is also Professor of Software Reliability at the Computing Laboratory, University of Kent, UK. He received a number of international prizes for geophysics in the 1970s and 80s, culminating in the 1987 Conrad Schlumberger prize for his work in computational geophysics. Shortly afterwards, he became interested in software reliability, and changed careers to study the design of high-integrity and safety-critical systems. In October 1998, he was voted amongst the leading international scholars of systems and software engineering for the period 1993-1997 by the *Journal of Systems and Software*. He holds a BA from King's College, Cambridge, and an MSc and PhD from the University of Manchester, all in mathematics; an ALCM in guitar from the London College of Music; and an LLM in IT law from the University of Strathclyde. Contact him at Oakwood Computing Associates and the Computing Laboratory, University of Kent, UK; lesh@oakcomp.co.uk; www.oakcomp.co.uk.